



**Original citation:**

Yaghi, A. A. (1983) The compilation of functional language into intensional logic. University of Warwick. Department of Computer Science. (Department of Computer Science Research Report). (Unpublished) CS-RR-056

**Permanent WRAP url:**

<http://wrap.warwick.ac.uk/60759>

**Copyright and reuse:**

The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

**A note on versions:**

The version presented in WRAP is the published version or, version of record, and may be cited as it appears here. For more information, please contact the WRAP Team at: [publications@warwick.ac.uk](mailto:publications@warwick.ac.uk)



<http://wrap.warwick.ac.uk/>

The University of Warwick

# THEORY OF COMPUTATION

REPORT NO. 56

The Compilation  
of a  
Functional Language  
into  
Intensional Logic

by

Ali AG Yaghi

Computer Science Department,  
University of Warwick,  
Coventry CV4 7AL  
England

December 1983

# The Compilation of a Functional Language into Intensional Logic

Ali AG Yaghi

Computer Science Dept.  
University of Warwick  
Coventry CV4 7AL  
ENGLAND

## ABSTRACT

In this paper we propose a new implementation technique for functional languages by compilation into intensional logic, [Car49] [Mon74]. Based on this logic, we define the concept of intensional algebras in which the value of a symbol in the signature is a family of objects which denotes its value at different worlds in the universe. Our target code is the family of languages DE, **definitional equations**; it is the family  $\{DE(A) \mid A \text{ is an intensional algebra}\}$ . A program in a member of this family is a set of non-ambiguous equations defining nullary variable symbols. One of these variables should be the symbol **result**.

To show, both the validity and the efficiency, of our approach we apply it to the family of languages lswum. The source family

lswum =  $\{lswum(B) \mid B \text{ is an extensional algebra}\}$  is the subset of Landin's lswim which does not allow higher order functions. Moreover, in lswum we allow lswim's where-clause only in defining functions; and we assume that the equations are non-ambiguous. The extension of our approach to cope with higher order functions is straightforward and is discussed in [Yag84].

Finally we give, briefly, two algorithms for evaluating intensional expressions. The first is **reductive** and based on a set of rewrite rules; the second is **dynamic** and based on a demand driven data flow architecture.

---

This work was supported by a fellowship from the Science and Engineering Research Council of the United Kingdom.

## Table of Contents:

- 0- Introduction
- 1- The Source Language ISWUM
- 2- The Target Language DE
- 3- Intensional Logic
- 4- Intensional Algebras
- 5- Example: The Algebra *FUN0* and Functions without  
Nullary globals
- 6- The Compilation of ISWUM, and the Algebra *Flo*
  - 6-1 Lists with Back Pointers
  - 6-2 The Intensional Algebra *Flo*
  - 6-3 Rewrite Rules for *Flo*
- 7- The Translation of ISWUM into DE
  - 7-1 The Translation Algorithm
  - 7-2 The Correctness of the Compilation.
- 8- Conclusion and Implementation
  - 8-1 The Reductive Approach
  - 8-2 The Demand Driven Approach
- 9- References

## Acknowledgments

I wish to thank my friend and supervisor W. Wadge for his continuous help and encouragement. Thanks are also due to Prof. D. Park for the many helpful comments he has provided during writing this paper.

## 0- Introduction:

In this paper we propose a new implementation technique for functional languages by compilation into intensional logic, [Mon74] [Car49]. This approach, we believe, will offer an efficient technique for implementing functional languages. It is not committed to a particular hardware, or to a particular evaluation technique; nevertheless it lends itself directly to demand driven tagged data flow architecture.

Intensional logic is the study of **indexical expressions**, that is "words and sentences of which the reference cannot be determined without the knowledge of the context of use" [Mon74]. Thus, if  $U$  is a collection of contexts and  $D$  is a domain then the intensional value of an expression is an indexed family denoting the 'extensional' value which the expression takes at different contexts. That is, the intensional value is a function in  $[U \rightarrow D]$ . We shall denote the latter by  ${}^U D$ . We call the set of contexts the universe, or the set, of possible worlds.

The main idea behind this approach is to translate programs of a first order language, like *Isyum*, into a nullary order equational language over intensional terms, like *DE*. This is achieved by introducing a collection of intensional operators for function application and binding, then remove all function definitions from the source program. For example, the function definition

$$F(X) = X * X$$

is translated into the nullary equation

$$F = X * X$$

Knowing that the value of the function  $F$  depends on the value which its formals take from a context to another, we propose the set of function invocation to be the universe of possible worlds. The value of  $F$ , then, at the world (or context)  $i$  should be the value of  $F$  at the world in which the formal  $X$  takes the actual value of the  $i$ 'th application of  $F$  in the source program.

Hence, for binding actuals to their formals we introduce the intensional operator **act**, where for a formal **Z** and expressions **a**, **b**, **c**, ...

$$Z = \text{act} (a, b, c, \dots)$$

denotes that the value which the formal **Z** takes is, **a** at the first call, **b** at the second call, **c** at the third, and so on ... . Moreover, we introduce the family of operators **call**, where  $\text{call}_i F$  denotes the *i*'th call of the non-nullary variable symbol **F**. Hence the equations

$$\begin{aligned} F(X) &= X * X \\ Y &= F(2) + F(3) \end{aligned}$$

can be translated into

$$\begin{aligned} F &= X * X \\ Y &= \text{call}_0 F + \text{call}_1 F \\ X &= \text{act} (2, 3) \end{aligned}$$

Let us assume, for the time being, that the universe is denoted by the set of lists of natural numbers. The head of a list denotes the present function call, while the tail denotes the sequence of calls which led to the present one. Consequently,

if we define the value of the expression  $\text{call}_i F$  at a list  $\varphi$

to be the value of **F** at the list  $\text{cons}(i, \varphi)$ ,

and the value of  $\text{act}(x_0, \dots, x_{n-1})$  at the list  $\vartheta$

to be the value of  $x(\text{hd}(\vartheta))$  at  $\text{tail}(\vartheta)$

then the value of  $\text{call}_1 F$  at a list  $\alpha$  will be equal to

the value of **F** at the list  $\text{cons}(1, \alpha)$

which is equivalent to the value of  $X * X$  at  $\text{cons}(1, \alpha)$

hence, as the value of **X** at  $\text{cons}(1, \alpha)$  is the value of **act (2,3)** at  $\text{cons}(1, \alpha)$

which is 3, then the value of **F** at  $\alpha$  is 9.

In section 1 of this paper, we introduce the source language **lswum**. It is worth mentioning here that, throughout our work, we follow the terminology of [Lan66], and [WASH83] and define a language to be a family in which each

member is uniquely determined by an algebra of data types. The language **Iswum** = {Iswum(B)|B is an extensional algebra} is the subset of Landin's Iswim which does not allow higher order functions. Moreover, in Iswum we allow Iswim's where-clause only in defining functions; and we assume that the equations are non-ambiguous. The extension of our approach to cope with higher order functions is straight forward and is discussed in [Yag83]. We chose Iswum because we believe that any lambda-based first order equational functional language can be translated , simply, into a member of this family.

In section 2, we give a brief and general view of intensional logic. We define the family  $L$  of intensional languages, then we introduce the concepts of intensional structure and intensional environment. Based on this logic, we define in section 3, an intensional algebra to be a triple  $\langle U, F, D \rangle$  where  $U$  is a universe of possible worlds,  $D$  is a domain of objects, and  $F$  is the intensional interpretation function which assigns meaning to different symbols of the signature. The intensional value of an  $n$ -ary symbol in the signature is a function in  $[(^U D)^n \rightarrow ^U D]$ . We call  $D$  the **extensional** domain of the algebra, and  $^U D$  the **intensional domain**.

We have proposed in the above example the set of lists of natural numbers as a universe for the algebra which interprets the symbols **call** and **act** , and hence function application and actuals binding. In section 3 however, we will show that such a universe is not sufficient for compiling Iswum. We propose, and define, the set of **lists with back pointers** over the natural numbers as a universe. An element in such a set is a list with two tails; a **dynamic** tail (or a calling tail), and a **static** tail (or a defining tail).

We define then the intensional algebraic function, or the family of intensional algebras  $Flo$ . For a continuous extensional (classical)  $\Sigma$ -algebra  $B$ ,  $Flo(B)$  is the continuous intensional  $\Sigma'$ -algebra where  $\Sigma'$  is  $\Sigma \cup \vartheta$  and  $\vartheta$  is the

set of constant symbols

$$\{\gamma, \text{act}, \text{call}\} \cup \{\text{gcall}, \text{ggcall}, \dots, \text{g}^n\text{call}\}$$

The universe of  $\text{Flo}(\mathbf{B})$  is the set of lists with back pointers over the naturals. We denote this by  $\text{bl}(\omega)$ . The intensional meaning of a constant symbol in  $\Sigma$ , given by the algebra  $\text{Flo}(\mathbf{B})$ , is the pointwise extension of its meaning in the algebra  $\mathbf{B}$ . The constant symbols of the set  $\mathfrak{V}$  are assigned some (non-pointwise) intensional operators; i.e functions in  ${}^{\mathbf{U}}\mathbf{D} \rightarrow {}^{\mathbf{U}}\mathbf{D}$ . These operators, as in the example above, will enable us to resolve some of the complexities associated with functional languages, like calling global, function calls, binding actuals, ...etc.

Our target code is the family of languages DE, for **definitional equations** and is defined in section 4. It is the family  $\{\text{DE}(\mathbf{A}) \mid \mathbf{A} \text{ is an intensional algebra}\}$ . A program in a member of this family is a set of non-ambiguous equations defining nullary variable symbols. One of these symbols should be **result**. For an extensional algebra  $\mathbf{B}$ , the source language  $\text{Iswum}(\mathbf{B})$  is compiled into  $\text{DE}(\text{Flo}(\mathbf{B}))$ . The compilation algorithm, described in section 5, is to translate the functional equational language  $\text{Iswum}(\mathbf{B})$  into the nullary equational language DE over the intensional terms of the algebra  $\text{Flo}(\mathbf{B})$ .

Finally, in section 6 we give, briefly, two algorithms for evaluating the expressions of Flo. The first is *reductive* and is based on the set of rewrite rules, described in section 3.3, for the algebra Flo. The second is *dynamic* and based on a demand driven data flow architecture.



## 1- The Source Language Iswum

The language Iswum  $\dagger$  is the subfamily of Landin's Iswim  $\dagger\dagger$  [Lan66] which does not allow higher order functions. In Iswum, also, we do not allow the **where** clause except in defining functions and as the outer casing for the program. Hence the following Iswim(Q)-program

```

F(2,X) where
  F(A,B) = A * A + 2 * B ;
  X = Z + V where
    V = 12 ;
    Z = V * V ;
  end;
end

```

is not in Iswum(Q) because the inner **where** clause is used to define the nullary variable symbol **X**. However, the expression

```

Z + V where
  V = 12 ;
  Z = V * V ;
end

```

is a program in Iswum(Q). Moreover, we assume that Iswum-programs are already named such that a variable symbol is used only once in the set of all symbols in the program. For example, the following is not a program in Iswum(Q) as the symbol **A** occurs as a local in the main clause and in the definition of **F**.

```

F(3)+A where
  F(X) = if X <= 1 then 1 else X * A fi
  where
    A = F(X-1) ;
  end;
  A = 4 ;
end

```

As Iswum is a family, then the syntax and the semantics of each member

---

$\dagger$  I See What U Mean  
 $\dagger\dagger$  I See What It Means

$\text{Iswum}(A)$  is uniquely determined by the algebra of data types  $A$ . That is why we decided, here, to give the abstract syntax of  $\text{Iswum}$  rather than using BNF equations.

### The Abstract Syntax of $\text{Iswum}$ :

Given an extensional  $\Sigma$ -algebra  $A$ , a program in  $\text{Iswum}(A)$  is a valid expression. A valid expression is either simple or a where-clause, and these are defined recursively as follows:

1- If  $\alpha$  is an  $n$ -ary constant symbol in  $\Sigma$  and

$x_0, \dots, x_{n-1}$  are simple expressions, then  $\alpha(x_0, \dots, x_{n-1})$   
is a simple expression.

2- If  $\vartheta$  is an  $n$ -ary variable symbol,  $x_0, \dots, x_{n-1}$  are  $n$  simple  
expressions, then  $\vartheta(x_0, \dots, x_{n-1})$  is a simple expression.

3- If  $\varepsilon$  is either a simple expression or a where-clause  
and  $\delta_0, \dots, \delta_{m-1}$  are  $m$  valid unambiguous  
definitions, then

$\varepsilon$  **where**

$\delta_0$

$\dots$

$\delta_{m-1}$

**end**

is a where-clause.

Where a valid definition is defined as

1- If  $\vartheta$  is a nullary variable symbol and  $\varepsilon$  is a simple  
expression. Then

$\vartheta = \varepsilon$  is a valid definition.

2- If  $\vartheta$  is a non-nullary variable symbol, and

$\eta_0, \dots, \eta_{n-1}$  are  $n$  nullary variable symbols,

$\varepsilon$  is either a simple expression or a where-clause. Then

$$\vartheta(\eta_0, \dots, \eta_{n-1}) = \varepsilon$$

is a valid equation.

We define the semantics of  $\text{Iswum}(A)$  to be that of  $\text{Iswim}$ , i.e. the meaning of a program in  $\text{Iswum}(A)$ , for an algebra  $A$ , is the same as its meaning according to the semantics of  $\text{Iswim}$ .

**Example:**

Let  $QS$  be the algebra which consists of the rationals together with the usual mathematical operations, and the set of strings. Then the following  $\text{Iswum}(QS)$ -program calculates the  $n$ 'th Fibonacci number, when  $n$  is positive; and returns the string **'Your input is negative'** for negative inputs.

```

fibb (N) where
  fibb(B) = if B >= 0 then fib(B)
             else 'Your input is negative' fi;
  fib(A) = if A <= 1 then A
            else fib(A-1) + fib(A-2) fi ;
end

```

## 2 The Target Language DE:

### *Introduction:*

We introduce here the family of programming languages DE, for **Definitional Equations**. It is the family

$$DE = \{ DE(A) \mid A \text{ is an algebra} \}.$$

Informally speaking, a program written in a member of DE is a set of compatible equations defining nullary variable symbols. Each equation is of the form **variable** = **expression**. The set is compatible means that each variable is defined at most once. One of these equations should define the symbol **result**. Since a program is a set, the order in which the definitions are written is not significant. So, neither function definitions nor lswim-like where-structures is allowed in DE.

Clearly, DE is a trivial equational language, however, its importance comes when we start considering the subfamily of DE in which the equations are over intensional terms. Hence, use the language as a target in our compilation technique.

Since DE is a family, then both the syntax and the semantics of a member  $DE(A)$  is determined by the algebra of data types A.

### *The Syntax of DE*

Let A be an algebra. Then the abstract syntax of  $DE(A)$  is defined as follows:

The set of  $DE(A)$ -expressions is the smallest set X such that

- All nullary variable symbols are in X.
- If  $\psi$  is an n-ary constant symbol in the signature of A,  
and  $\xi_0, \dots, \xi_{n-1}$  are in X then  
 $\psi(\xi_0, \dots, \xi_{n-1})$  is in X

A  $DE(A)$ -equation consists of a left hand side (lhs) which is a nullary variable symbol and a right hand side (rhs) which is a  $DE(A)$ -expression

A  $DE(A)$ -program is a set of  $DE(A)$ -equations. such that

- \*- Compatibility: each lhs variable symbol is defined at most once.
- \*- One of these equations defines the symbol **result**,

*Occurrences of Variables in DE:*

Clearly, occurrences of variable symbols in  $DE(A)$ -expressions are all free. In a  $DE(A)$ -program however, an occurrence of a variable symbol is bound if that symbol occurs as a lhs of an equation in the program; otherwise it is a free occurrence.

### The semantics of DE

#### $DE(A)$ -expressions:

Given an algebra  $A$ , and an environment  $\varepsilon$ ; the semantics of a  $DE(A)$ -expression  $\tau$ , denoted by  $\models_{A,\varepsilon} \tau$  is defined recursively as follows:

- a: If  $\tau$  is a nullary variable symbol, then the value of  $\tau$  in the environment  $\varepsilon$  is  $\varepsilon(\tau)$ ; i.e. the value assigned to  $\tau$  by the environment  $\varepsilon$ .

$$\models_{A,\varepsilon} \tau = \varepsilon(\tau)$$

- b: If  $\tau$  is of the form  $\psi(x_0, \dots, x_{n-1})$  where  $\psi$  is an  $n$ -ary constant symbol, and  $x_0, \dots, x_{n-1}$  are  $DE(A)$ -terms, then the value of  $\tau$  in the environment

$\varepsilon$  is the value of  $\psi$  assigned to it by the algebra

$A$  applied to the values of  $x_0, \dots, x_{n-1}$  relative to the algebra  $A$   
and the environment  $\varepsilon$ . i.e.

$$\models_{A,\varepsilon} \tau = A(\psi) (\models_{A,\varepsilon} x_0, \dots, \models_{A,\varepsilon} x_{n-1})$$

*Definition:*

Given an environment  $\varepsilon$ , and a  $DE(A)$ -equation

**V = Expression**

We say that  $\varepsilon$  satisfies the equation if and only if

$$\varepsilon(V) = \models_{A,\varepsilon} (\text{Expression}).$$

We say that  $\varepsilon$  satisfies the  $DE(A)$ -program  $P$  if and only if  $\varepsilon$  satisfies all the equations of  $P$ .

*$DE(A)$ -Programs:*

Given an algebra  $A$ , the value of a  $DE(A)$ -program  $P$  in the environment  $\varepsilon$ , is the value of the variable *result* in the least environment  $\varepsilon'$ , such that  $\varepsilon'$  satisfies the equations of  $P$  and agrees with  $\varepsilon$  except at most on the values assigned to the locals of  $P$ .

### 3 Intensional Logic:

Both logicians and formal linguists recognize two theories in the study of meaning of languages. The theory of *reference* or *denotation* and the theory of *meaning* or *pragmatics*. The first was developed by Tarski, Goedel, Hilbert and others; and was later known as the model theoretic approach to semantics. This is concerned with the relation between expressions and the objects they denote or refer to, according to a prior denotation function. On the other hand, *pragmatics* is the study of the relation between expressions, the objects they denote, and the contexts of use or utterance. In other words, it is concerned with the study of *indexical expressions*, that is "words and sentences of which the reference cannot be determined without knowledge of the context of use" [Mon74]. Hence, if we assume that  $U$  is a set of contexts, and  $D$  is a domain then the value of an expression is a function from  $U$  to  $D$ , i.e. an element in  ${}^U D$ .

To cast some light on intensional logic and pragmatics we define here a simple family  $L$  of intensional languages. In members of  $L$ , we shall consider  $L$ -formulas as  $L$ -terms. So in the interpretation of a language we shall talk about values of terms rather than validity and satisfaction of formulas. This in some sense coincides with our intended use of the language as an implementation technique rather than as a proof system. Moreover,  $L$  is a family of first order languages, i.e. we allow first order intensional operators only.

#### *Syntax :*

Let  $\Sigma$  be a set of constant symbols with different arities, then the set of terms of  $L(\Sigma)$  is defined recursively as follows:

- If  $\psi$  is an  $n$ -ary constant symbol in the signature  $\Sigma$ , and

$\xi_0, \xi_1, \dots, \xi_{n-1}$  are  $n$  terms,

then

$\psi(\xi_0, \xi_1, \dots, \xi_{n-1})$  is a term.

- If  $\vartheta$  is an  $n$ -ary variable symbol, and

$\xi_0, \xi_1, \dots, \xi_{n-1}$  are  $n$  terms,

then

$\vartheta(\xi_0, \xi_1, \dots, \xi_{n-1})$  is a term.

*Semantics:*

We give here a formal definition of an *intensional structure* in which we define, as well, the *intension* function relative to the set of possible worlds of the structure. Then we define the *meaning* function relative to an intensional structure together with an intensional environment. It is worth noting here that an intensional environment must be defined relative to a certain structure so that it maps each variable symbol to an element of the intensional domain of the structure; that is, a function from the set of possible worlds of the structure to the extensional domain.

*Definition 1:*

Let  $L(\Sigma)$  be an intensional language. An *intensional interpretation*, or a structure, for  $L$  is a triple  $\langle U, F, D \rangle$  where

$U$  is a nonempty set of possible worlds, the **universe**

$D$  is a nonempty domain of objects, the **extensional domain**,

$F$  is the **intension function** which assigns to every

$n$ -ary constant symbol  $\psi$ , in the signature  $\Sigma$  a

function in

$$[(^U D)^n \rightarrow {}^U D]$$

*Definition 2:*

Let  $U (= \langle U, D, F \rangle)$  be an intensional structure.



A  $U$ -intensional environment  $\varepsilon$  is a function

$$[ V \rightarrow {}^U D ] \text{ where } V \text{ is the set of variable symbols.}$$

Hence, an intensional environment assigns to each nullary variable symbol a function in  ${}^U D$

*Definition 3:*

Let  $L(\Sigma)$  be an intensional language.

Let  $U (= \langle U, D, F \rangle)$  be an intensional structure for  $L(\Sigma)$ .

Let  $\varepsilon$  be a  $U$ -intensional environment.

Then the *intension* of an  $L(\Sigma)$ -term  $\tau$  relative to the structure  $U$  and the  $U$ -intensional environment  $\varepsilon$  is denoted by  $\models_{U,\varepsilon}(\tau)$  and is defined recursively as follows:

If  $\tau$  is of the form  $\psi(\xi_0, \xi_1, \dots, \xi_{n-1})$

where  $\psi$  is an  $n$ -ary constant symbol and

$\xi_0, \xi_1, \dots, \xi_{n-1}$  are  $n$  terms then

$$(\models_{U,\varepsilon} \tau) = F(\psi)(\models_{U,\varepsilon} \xi_0, \dots, \models_{U,\varepsilon} \xi_{n-1})$$

If  $\tau$  is a nullary variable symbol, then

$$(\models_{U,\varepsilon} \tau) = \varepsilon(\tau).$$

#### 4- Intensional Algebras:

Before introducing the concept of intensional algebras, we shall recall some definitions of classical algebras- or what we shall call from now on extensional algebras. For more details on these algebras we refer the reader to [ADJ78].

*Definitions 1:*

A signature, or an operator domain,  $\Sigma$  is a collection of constant symbols

with different arities.

An extensional  $\Sigma$ -algebra  $A$  is an ordered pair  $\langle F, D \rangle$  such that;  $D$  is a non-empty domain, and  $F$  is a function mapping each  $n$ -ary constant symbol in  $\Sigma$  to an operation of degree  $n$  on the domain  $D$ ; i.e. a function in  $[D^n \rightarrow D]$

If  $A (= \langle F, D \rangle)$  is an extensional  $\Sigma$ -algebra, then  $D$  is called the domain of  $A$  and denoted by  $|A|$ .  $\Sigma$  is called the signature of  $A$ .

A domain is a CPO which, at least, has the truth values  $\{tt, ff\}$  and Scott's undefined element  $\perp$ .

*Definition 2:*

An intensional  $\Sigma$ -algebra  $B$  **upon**  $U$  is a triple  $\langle F, D, U \rangle$  such that;  $D$  is a non-empty domain,  $U$  is a non-empty set; and  $F$  is a function mapping each  $n$ -ary constant symbol in  $\Sigma$  to a function in  $[({}^U D)^n \rightarrow {}^U D]$ .

If  $A (= \langle F, D, U \rangle)$  is an intensional  $\Sigma$ -algebra, then  $D$  and  $U$  are called, respectively, the **extensional domain** and the **universe** of  $A$ . The set of all functions from the universe  $U$  to the extensional domain  $D$ , denoted by  ${}^U D$ , is called the **intensional domain** of  $A$  and denoted by  $|A|$ . We can say, then, that an intensional  $\Sigma$ -algebra  $A$  maps each  $n$ -ary operator symbol in the signature  $\Sigma$  to a function of degree  $n$  on its intensional domain. Hence, when we talk about the domain of an intensional algebra, from now on, we mean the intensional domain.

*Definition 3:*

For any two (either extensional or intensional exclusively) algebras  $A$  and  $B$ ,  $A$  is said to be a *subalgebra* of  $B$ , denoted by  $A \subset B$ , if

- the signature of  $A$  is a subset of the signature of  $B$ , and
- the algebra  $B$  assigns to the constant symbols of the signature of  $A$  the same functions as those assigned by  $A$ ; i.e.

$B / \text{signature of } A = A$

Or in other words, for every constant symbol  $\psi$  in the signature of  $A$ ,

$$\psi_B = \psi_A$$

*Definition 4:*

Let  $A$  and  $B$  be two  $\Sigma$ -algebras, then a  $\Sigma$ -homomorphism  $f: A \rightarrow B$  is a function such that for any  $n$ -ary operation symbol  $\psi$  in the signature of  $A$  and any  $a_0, \dots, a_{n-1}$  in  $|A|$

$$f(A(\psi)(a_0, \dots, a_{n-1})) = (B(\psi))(f(a_0), \dots, f(a_{n-1}))$$

*Definitions 5:*

Let  $l_\Sigma$  be the identity function for composition in the class of  $\Sigma$ -homomorphisms. That is, for any  $\Sigma$ -homomorphism  $f$

$$f \circ l_\Sigma = l_\Sigma \circ f = f$$

Then  $g$  is called the inverse of  $f$ , and denoted by  $f^{-1}$ , if and only if

$$g \circ f = f \circ g = l_\Sigma$$

Let  $h$  be a  $\Sigma$ -homomorphism. Then  $h$  is called a  $\Sigma$ -isomorphism iff there is a  $\Sigma$ -homomorphism  $g$  such that  $g = h^{-1}$ . Any two  $\Sigma$ -algebras  $A$  and  $B$  are isomorphic, written as  $A \equiv B$  iff there exist a  $\Sigma$ -isomorphism from  $A$  to  $B$ .

*Proposition 1*

If the universe of an intensional  $\Sigma$ -algebra  $A$  is a singleton, then there exist an extensional  $\Sigma$ -algebra  $B$  such that

$$A \equiv B$$

*Definition 6:*

Let  $A$  be an extensional  $\Sigma$ -algebra, and  $U$  be a set. The *pointwise extension* of  $A$  **upon**  $U$  is the intensional algebra, denoted by  $A^U$ , and defined as follows:

for every  $n$ -ary operator symbol  $\vartheta$  in the signature  $\Sigma$ ,  $A^U$  assigns a function of degree  $n$  on the set  ${}^U|A|$  such that

$$\begin{aligned} &\forall i \in U, \text{ and } \forall \xi_0, \dots, \xi_{n-1} \text{ in } {}^U|A| \\ &A^U(\vartheta(\xi_0, \dots, \xi_{n-1}))(i) = A(\vartheta(\xi_0(i), \dots, \xi_{n-1}(i))) \end{aligned}$$

An intensional algebra  $B$  is said to be **based on** the pointwise intensional algebra  $A$  if  $A$  is a subalgebra of  $B$ .

### 5 Example: The Algebra *FUN0* and Functions without Nullary Globals:

In this section we give a preliminary example of an intensional algebra, and show how to employ such a concept in compiling functions. We consider here, only, the subset of *lswum* where function definitions do not have global occurrences of nullary variable symbols. Consider the following example

```

F(X) where
  X = F(2) + G(3) + G(2) ;
  F(A) = A * A ;
  G(B) = 2 * B ;
end

```

Ideally, what we would like to be able to say instead of  $F(A) = A * A$  is  $F = A * A$ . Since the value of  $F$  is a function of the value of its formal  $A$ , and the value which  $A$  takes varies from a call to another; then we can consider the universe of the algebra to be the set of function calls. Thus, from the equation  $F = A * A$ , the value of  $F$  in any world is dependent on the value which  $A$  takes in that same world. We shall represent the universe of function calls by the set of lists of natural numbers. The head of a list denotes the present (the current) function call, while the tail denotes the list of calls which led to the present one.

From the text of the program above, we know that the actual of the first call of  $F$  is  $X$ , and of the second call is  $2$ . For this, we introduce the operator **act** and define the formal  $A$  of  $F$  to be

$$A = \text{act}(X, 2)$$

This equation, informally, states that the value of  $A$  at the first call is the value of  $X$ , and at the second call is the value of  $2$ . For function application, we introduce the family of operator symbols **call** where  $\text{call}_i H$  denotes the  $i$ 'th application of the function  $H$ . Hence, the expressions  $F(X)$  and  $F(2)$  are translated, resp., into  $\text{call}_0 F$  and  $\text{call}_1 F$ . Similarly,  $G$  has been called twice in the program, hence

$$G = 2 * B$$

$B = \text{act } (3, 2)$

and  $G(3)$ ,  $G(2)$  are translated, resp., into  $\text{call}_0 G$  and  $\text{call}_1 G$ . So, the above program is translated into

```

result =  $\text{call}_0 F$ 
X =  $\text{call}_1 F + \text{call}_0 G + \text{call}_1 G$  ;
F =  $A * A$  ;
A =  $\text{act } (X, 2)$  ;
G =  $2 * B$  ;
B =  $\text{act } (3, 2)$ 

```

To formalize the discussion above we introduce the family of intensional algebras  $FUN0$ . It is a function which maps extensional algebras to intensional ones. For an extensional  $\Sigma$ -algebra  $A$ ,  $FUN0(A)$  is an intensional  $\Sigma'$ -algebra. This is because  $FUN0(A)$ , besides assigning meaning to the constant symbols of  $A$ , it assigns meaning to the new constant symbol **act** and to the family of symbols **call**. However, we can, simply, say that  $FUN0(A)$  is a  $\Sigma$ -algebra because  $\Sigma'$  is a function of  $\Sigma$ . It is the union of  $\Sigma$  and the set  $\{\text{act}, \text{call}\}$ .

### The Intensional Algebra $FUN0$

Given an extensional  $\Sigma$ -algebra  $A$ , the intensional  $\Sigma$ -algebra  $FUN0(A)$  is the least  $\dagger$  intensional  $\Sigma'$ -algebra such that:

The signature  $\Sigma'$  of  $FUN0(A) = \Sigma \cup \{\text{call}_i : i \in \omega\} \cup \{\text{act}\}$

The universe  $L$  of  $FUN0(A)$  is the set of all lists of natural numbers.

The extensional domain of  $FUN0(A)$  is the domain of  $A$ .

---

$\dagger$  the least up to the relation *subalgebra* defined before

*FUN0* extends **A** pointwise upon the universe *L*, that is

- for every *n*-ary constant symbol  $\psi$  in the signature of **A**,
- for every *n* terms  $\xi_0, \dots, \xi_{n-1}$  in *FUN0*(**A**),
- and for every list  $\varphi$  in the universe *L*,

$$(\text{FUN0}(\mathbf{A})(\psi(\xi_0, \dots, \xi_{n-1})))_{\varphi} = \mathbf{A}(\psi)((\text{FUN0}(\xi_0))_{\varphi}, \dots, (\text{FUN0}(\xi_{n-1}))_{\varphi})$$

*FUN0* assigns meaning to the family of constant symbols **call** as follows:

- for every  $i \in \omega$
- for every variable symbol  $\xi$ , and
- for every list  $\varphi$  in the universe *L*

$$(\text{FUN0}(\mathbf{A})(\text{call}_i(\xi)))_{\varphi} = \xi_{\text{cons}(i, \varphi)}$$

*FUN0* assigns the following meaning to the constant symbol **act**

- for every list  $\varphi$  in the universe *L*
- and for every *n* expressions  $\xi_0, \dots, \xi_{n-1}$

$$(\text{FUN0}(\mathbf{A})(\text{act}(\xi_0, \dots, \xi_{n-1})))_{\varphi} = (\xi(\text{hd}(\varphi)))_{\text{tl}(\varphi)}$$

- where *hd*, *tl*, and *cons* are the usual head, tail, and construct functions on lists (resp.).

According to the analysis above and the definition of *FUN0*, we can compile the subset of *lswum* where function definitions do not have global occurrences of nullary variable symbols to equations over terms of the intensional algebra *FUN0*. In other words, if we call such a subset *lswum0*, then for an extensional  $\Sigma$ -algebra **A**, the subset *lswum0*(**A**) can be compiled into the member *DE*(*FUN0*(**A**)). The latter of course is a member in our target language *DE*. In the following example, we show that we can compile and evaluate even recursive functions to equations over the algebra *FUN0* as far as the definition of the

function does not have global occurrences of nullary variable symbols.

**Example :**

Consider the following program in  $\text{Iswum}(\mathbb{Z})$ , where  $\mathbb{Z}$  is the algebra of integers together with the usual mathematical operations  $+$ ,  $-$ ,  $*$ , and  $\text{div}$ .

```

Fac (2) where
  Fac(X) = if X le 1 then 1 else X * Fac(X-1) fi;
end

```

Using the same analysis described above, we can translate this program into the following program in  $\text{DE}(\text{FUN0}(\mathbb{Z}))$

```

result = call0Fac;

Fac = if X le 1 then 1 else X * call1Fac fi;

X = act (2,X-1);

```

Note that, the value of the DE-program should be the value of **result** at the empty list because no functions had been called yet.

The value of **result** at the list  $[]$  equals

the value of **call**<sub>0</sub> **Fac** at  $[]$  (by substitution)

which is equal to the value of **Fac** at the list  $[0]$

(by the interpretation of **call**)

by direct substitution, this is equal to the value of

(if X le 1 then 1 else X \* **call**<sub>1</sub>**Fac** fi) at the list  $[0]$

By substitution, the value of **X** at  $[0]$  is equal to the value of

**act** (2,X-1) at  $[0]$  which is 2 (by the interpretation of **act**)

Thus the value of **result** at  $[0]$  equals the value of

( 2 \* **call**<sub>1</sub>**Fac**) at  $[0]$  which equals to the value of

2\* (the value of **Fac** at the list  $[1\ 0]$ )

(note here that both **\*** and **2** are interpreted pointwise)

this is equal to 2\*( the value of if X le 1 then 1 else X \* **call**<sub>1</sub> **Fac** fi) at  $[1\ 0]$ )

now the value of **X** at  $[1\ 0]$  is equal to the value of



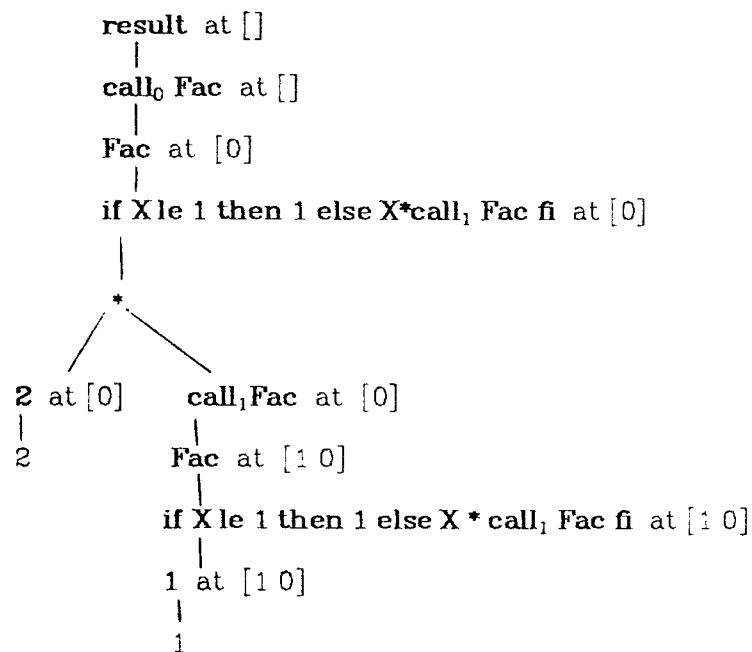
**act** (2,X-1) at [1 0] (by substitution)

which is the value of **X-1** at [0] (by the definition of **act**)

which is 1 because the value of **X** at the list [0] is 2

Hence, the value of **result** is 2\*1.

The above evaluation can be represented in the following tree where lines denotes equalities:



## 6 The Compilation of lswum and the Algebra Flo

Once we start considering lswum-programs and allow function definitions to have global occurrences of nullary symbols, the algebra *FUN0* fails to interpret either function calls or globals. For example, consider the following lswum(Z)-program:

```

Y where
  Y = F(2);
  A = 5 ;
  F(a) = a * A + G(A);
  G(b) = A * b ;
end

```

If we translate  $F(a)$  and  $G(b)$  resp. as

$$F = a * A + \text{call}_0 G \quad \text{and} \quad G = A * b$$

then the value of  $F(2)$  is the value of the expression

$a * A + \text{call}_0 G$  at the list  $[0]$  as  $F(2)$  is the first call of  $F$ . This depends on the value of  $A$  at  $[0]$ . However, we know that  $A$  is not defined within any function call. It is defined at the outer most level (**where**-clause) but called from the first call of  $F$ . That is, the value of  $A$  at  $[0]$  should be undefined, and we should evaluate  $A$  at  $[]$  instead. Moreover, the value of  $F$  at  $[0]$  depends on the value of  $\text{call}_0 G$  which is equal to the value of  $G$  at the list  $[0\ 0]$ ; this depends on the value of  $A$  at  $[0\ 0]$  and the value of  $A$  at  $[0]$  which are both undefined. Such errors occur because the universe of *FUN0* denote lists of function calls, hence facilitating dynamic binding only. We need an algebra and a universe which facilitate for static binding as well as dynamic binding.

For this purpose, we define the intensional algebra *Flo*. The universe of *Flo*, is the set of special kind of lists. We call them **lists with back pointers**. The important property about these lists that they have two tails; the **dynamic** (or **calling**) **tail** and the **static** (or **defining**) **tail**.

We introduce now a formal definition of the set of b-lists (lists with back

pointers) together with the partial order **initial segment** defined on this set. We also define on the elements of this set the operations **dtl**, **stl** and **link** for the dynamic (calling) tail, the static (defining) tail and b-list constructions respectively. Next, we introduce a formal definition of the algebra Flo.

### 6.1 Lists with Back Pointers:

#### Definition:

Let  $A$  be a set. We define the set  $bl(A)$  of b-lists, or **lists with back pointers over the set  $A$** , together with the relation **initial segment**, denoted by  $\prec$  on the elements of  $bl(A)$  as follows:

1- the special symbol  $\Lambda$  is in  $bl(A)$ , and

$$\forall \alpha \in bl(A) \quad \Lambda \prec \alpha$$

2- if  $a \in A, c, d \in bl(A)$  then

$$\alpha = \langle a, c, d \rangle \in bl(A) \text{ whenever } d \prec c.$$

3-  $\forall \alpha = \langle a, c, d \rangle$  and  $\beta = \langle a', c', d' \rangle$  in  $bl(A)$

$$\alpha \subset \beta \Leftrightarrow \alpha = \beta \text{ or } \alpha \prec d'$$

#### Definition:

We define the following functions on the elements of  $bl(A)$ ,

for any  $\alpha = \langle a, c, d \rangle \in bl(A)$

$$hd(\alpha) = a \quad hd(\Lambda) = \perp$$

$$dtl(\alpha) = c \quad dtl(\Lambda) = \perp$$

$$stl(\alpha) = d \quad stl(\Lambda) = \perp$$

#### Definition:

We define here the b-lists constructor **link**, which takes an element  $a \in A$ , and two lists  $\alpha, \beta \in bl(A)$ , where  $\beta \prec \alpha$ , and constructs a new b-list whose head is

$a$ ,  $\text{dtl}$  is  $\alpha$ , and  $\text{stl}$  is  $\beta$ .

For any  $a \in A$ , and  $\alpha, \beta \in \text{bl}(A)$  where  $\beta \prec \alpha$ , the b-list  $\text{link}(a, \alpha, \beta)$  is the b-list which satisfies the following equalities

$$\text{hd}(\text{link}(a, \alpha, \beta)) = a$$

$$\text{dtl}(\text{link}(a, \alpha, \beta)) = \alpha$$

$$\text{stl}(\text{link}(a, \alpha, \beta)) = \beta$$

## 6.2 The Intensional Algebra *Flo*

**Definition:**

Let  $A (= \langle F, D \rangle)$  be an extensional  $\Sigma$ -algebra. The intensional  $\Sigma$ -algebra  $\text{Flo}(A)$  is the triple  $\langle U, F', D \rangle$  where

a: the universe of possible worlds  $U$  is the set b-lists of natural numbers; i.e  $\text{bl}(\omega)$

b: the intensional interpretation function  $F'$  extends  $F$  pointwise.

That is, for every  $n$ -ary constant symbol  $\psi$  in  $\Sigma$ , the sequence of *Flo*-expressions  $x_0, \dots, x_{n-1}$ , and every  $u \in U$

$$(F'(\psi)(x_0, \dots, x_{n-1}))_u = F(\psi)(F'(x_0)(u), \dots, F'(x_{n-1})(u))$$

c: The function  $F'$  assigns meaning to the followings:

The symbol  $\gamma$ :

For every expression  $\varepsilon$ , and every  $u \in U$

$$(F'(\gamma \varepsilon))_u = \varepsilon_{\text{stl}(u)}$$

The symbol **act**

For every  $u \in U$ , and every sequence of

expressions  $x_0, \dots, x_{n-1}$

$$F'(\mathbf{act}(x_0, \dots, x_{n-1}))_u = x(\text{hd}(u))_{\text{dl}(u)}$$

The family of operator symbols  $\{\mathbf{call}_i\}_{i \in \omega}$

For every *Flo*-expression  $\varepsilon$

for every  $i \in \omega$ , and for every  $u \in U$

$$(F'(\mathbf{call}_i \varepsilon))_u = \varepsilon_{\text{link}(i, u, u)}$$

The family of operator symbols  $\{\mathbf{gcall}_i\}_{i \in \omega}$

For every expression  $\varepsilon$ , every  $i \in \omega$

and every  $u \in U$

$$(F'(\mathbf{gcall}_i \varepsilon))_u = \varepsilon_{\text{link}(i, u, \text{stl}(u))}$$

The family of operator symbols  $\{\mathbf{ggcall}_i\}_{i \in \omega}$

For every expression  $\varepsilon$ , every  $i \in \omega$ ,

and for every  $u \in U$

$$(F'(\mathbf{ggcall}_i \varepsilon))_u = \varepsilon_{\text{link}(i, u, \text{stl}(\text{stl}(u)))}$$

In general, for every  $j \in \omega$ , there is a family of operator symbols  $\{\mathbf{g}^j\mathbf{call}_i\}_{i \in \omega}$  such that

For every  $j \in \omega$ , for every expression  $\varepsilon$  in *Flo*( $A$ )

every  $i \in \omega$ ,

and every  $u \in U$

$$(F'(\mathbf{g}^j\mathbf{call}_i \varepsilon))_u = \varepsilon_{\text{link}(i, u, \text{stl}(\text{stl}(\dots \text{stl}(u) \dots))}$$

where  $\text{stl}$  is applied here  $j$  times.

### 6.3 Rewrite Rules for the algebra *Flo*:

These are **directed** equations for syntactic manipulation of terms. The difference between usual equations and rewrite rules or directed equations is

that equations denote symmetric equality, i.e.  $A = B$  implies  $B = A$ ; while in rewrite rules the equality is directional or the equality implication is one sided. Moreover, these rules are purely syntactic, and the only substitution allowed here is the one based on pattern matching. We shall, nevertheless, use the semantics to prove their correctness.

These rules are over the terms of  $Flo(A)$ , for any extensional algebra  $A$ ; which means that both sides of each rule are  $Flo(A)$ -terms. Thus proving a rule correct is to prove that both sides of the rule denote the same object (have the same meaning). Since  $Flo(A)$  is intensional, and the value of a term is a family denoting its value at each world in the universe, then we have to show that the equality holds for all the worlds in the universe. For example, to prove the rewrite rule  $A=B$ , for the algebra  $Flo(A)$  with universe  $U$ , we have to prove that

For every world  $u \in U$

$$(Flo(A)(A))_u = (Flo(A)(B))_u$$

Since  $Flo$  is a family of algebras, then the rewrite rules for each member,  $Flo(A)$ , is determined by the rewrite rules of both  $Flo$  and  $A$ . However, what we are going to introduce here is the set of rewrite rules for the whole family  $Flo$  no matter what  $A$  is. The only rule which concerns the algebra  $A$  is the first one. It states that the operators of  $Flo$  are distributive over those of  $A$ . For example, in  $Flo(Q)$ , where  $Q$  is the extensional algebra over the rationals, the following equation holds

$$call_i (X + Y - 3) = call_i X + call_i Y - call_i 3$$

We give now the rewrite rules for the algebra  $Flo$ , then justify the correctness of rules 1 and 3 as examples. Such a justification is based on the semantics of terms given by the definition of  $Flo$ . For the correctness of the other rules see [Yag83.1].

(Rule 0) If  $\psi$  is an  $n$ -ary operator symbol in the signature of  $A$ ,

and  $\mathbf{x}_0, \dots, \mathbf{x}_{n-1}$  are  $Flo(A)$ -expressions, then

for every operator symbol  $\vartheta$  in the signature of  $Flo$

$$\vartheta(\psi(\mathbf{x}_0, \dots, \mathbf{x}_{n-1})) = \psi(\vartheta(\mathbf{x}_0), \dots, \vartheta(\mathbf{x}_{n-1}))$$

If  $\mathbf{X}, \mathbf{X}_0, \dots, \mathbf{X}_{n-1}$  are  $Flo(A)$ -terms, then

$$\text{(Rule 1)} \quad \mathbf{call}_i(\mathbf{act}(\mathbf{X}_0, \dots, \mathbf{X}_{n-1})) = \mathbf{X}_i$$

$$\text{(Rule 2)} \quad \mathbf{gcall}_i(\mathbf{act}(\mathbf{X}_0, \dots, \mathbf{X}_{n-1})) = \mathbf{X}_i$$

$$\text{(Rule 3)} \quad \mathbf{call}_i(\gamma\mathbf{X}) = \mathbf{X}$$

(Rule 4) This is a family of rules concerning the family  $\mathbf{g}^n\mathbf{call}$ .

They are:

$$\mathbf{gcall}_i(\gamma\mathbf{X}) = \gamma\mathbf{X}$$

$$\mathbf{ggcall}_i(\gamma\mathbf{X}) = \gamma\gamma\mathbf{X}$$

$$\mathbf{gggcall}_i(\gamma\mathbf{X}) = \gamma\gamma\gamma\mathbf{X}$$

generally speaking

$$\mathbf{g}^r\mathbf{call}_i(\gamma\mathbf{X}) = \gamma^r\mathbf{X}$$

Proof: For simplicity, we shall denote  $(Flo(A)(\mathbf{X}))_u$  by  $[\mathbf{X}]_u$ .

Assume that  $U$  is the universe of  $Flo(A)$

$\mathbf{X}, \mathbf{X}_0, \dots, \mathbf{X}_{n-1}$  are  $Flo(A)$ -terms,

then for every  $u \in U$

$$\begin{aligned} (1) \quad & [\mathbf{call}_i(\mathbf{act}(\mathbf{X}_0, \dots, \mathbf{X}_{n-1}))]_u \\ &= [\mathbf{act}(\mathbf{X}_0, \dots, \mathbf{X}_{n-1})]_{\text{link}(i,u,u)} \\ &= [\mathbf{X}_i]_{\text{dnl}(\text{link}(i,u,u))} \\ &= [\mathbf{X}_i]_u \end{aligned}$$

$$\begin{aligned} (3) \quad & [\mathbf{call}_i(\gamma\mathbf{X})]_u = [\gamma\mathbf{X}]_{\text{link}(i,u,u)} \\ &= [\mathbf{X}]_{\text{stl}(\text{link}(i,u,u))} \\ &= [\mathbf{X}]_u \end{aligned}$$

*Example:* From the above rules we can derive many others. For example

$$\mathbf{call}_i (\mathbf{gcall}_j (\gamma X) = X$$

$$\mathbf{call}_i (\mathbf{call}_j (\mathbf{ggcall}_k (\gamma X) = X$$



## 7 The Translation of Iswum into DE

While a program in Iswum is either a simple expression or a where-clause, a program in DE is a set of equations. Hence, the translation algorithm will be a function mapping each Iswum-program into a set of equations in  $DE(Flo(A))$ . The compatibility required in the target DE-program is captured by the compatibility of the source Iswum-program. Syntactically, apart from being a simple expression, an Iswum-program can be a where-clause with a structured set of equations. That is, a where-clause which contains another clause as a submodule. We shall call a set of equations enclosed in a where-clause a **textual level**, so the program

```

X+C where
  X = Y(2,Z) ;
  Y(a,b) = V where
    ...
  end;
end

```

consists of two textual levels. The main (outer) one and the one defining the function symbol  $Y$ . Textually, the latter is **contained in** the former. We say *contained in* rather than a *subset of* because the latter means something in set-theoretic terms which does not agree with the scope conventions.

The important point we want to remark here is that translation (compilation) will never be done in a vacuum. When we compile an expression, we have to compile it relative to the set of equations, or the textual level, it appears in together with all the textual levels which contain the present one. This is so that we can determine whether the variable symbols which occur in the expression are locals or globals; moreover, we want to be able bind the variable symbols to their definitions properly. Therefore, we shall talk about the compilation of an expression  $\varepsilon$  relative to a textual level  $C$ , we shall denote this by  $comp_C(\varepsilon)$ . Moreover, a variable symbol  $\mathbf{v}$  is local in a textual level  $C$  if one of

the equations of  $C$  defines  $\mathbf{v}$ . We denote that by  $\text{local}_C(\mathbf{v})$ . Otherwise, it is  $\text{global}_C(\mathbf{v})$ . Moreover, we shall denote the catenation operator on strings by  $\wedge$ . For example, for the strings  $xyz$  and  $abc$ ,  $xyz \wedge abc$  is the string  $xyzabc$ .

### 7-1 The Translation Algorithm

Given a program  $P$  in  $\text{Iswum}(A)$ , for an extensional algebra  $A$ , the target program  $P'$  of  $\text{DE}(\text{Flo}(A))$  is defined recursively as follows:

if  $P$  is a simple expression in  $\text{Iswum}(A)$  then  $P'$  is the singleton

$$\{ \text{result} = \text{compexp}_\varphi P \}$$

if  $P$  is of the form

$$X \text{ where } D \text{ end}$$

where  $X$  is an expression and  $D$  is a set of equations

then  $P'$  is

$$\{ \text{result} = \text{compexp}_D(X) \} \cup \{ \text{compdef}_D(d) \mid d \in D \}$$

where for a textual level  $C$ , a variable symbol  $\mathbf{v}$ , a definition  $\mathbf{d}$  and an expression

$\varepsilon$

$$\text{compexp}_C(\varepsilon) =$$

if  $\varepsilon$  is of the form  $\text{op}(\mathbf{x}_0, \dots, \mathbf{x}_{n-1})$

where  $\text{op}$  is an  $n$ -ary constant symbol and

$\mathbf{x}_0, \dots, \mathbf{x}_{n-1}$  are  $n$  expressions,

then

$$\text{op}(\text{compexp}_C(\mathbf{x}_0), \dots, \text{compexp}_C(\mathbf{x}_{n-1}))$$

if  $\varepsilon$  is a nullary variable symbol then

$$\text{compvar}_C(\varepsilon)$$

if  $\varepsilon$  is a formal then  $\varepsilon$

if  $\varepsilon$  is of the form  $F(\mathbf{x}_0, \dots, \mathbf{x}_{n-1})$  where  $F$  is an

n-ary variable symbol and  $x_0, \dots, x_{n-1}$

are n expressions, then

$\text{compfunc}_C(F)$

$\text{compvar}_C(v) =$

if  $C = \Phi$  then  $v$

else if  $\text{global}_C(v)$  then

$\gamma(\text{compvar}_{C'}(v))$

where  $C'$  is the first outer textual level containing  $C$ .

else  $v$ .

$\text{compfunc}_C(F) =$

if  $C = \Phi$  then  $F$

else if  $\text{global}_C(F)$  then  $g \sim (\text{compfunc}_{C'}(F))$

else  $\text{call}_i F$ .

where  $C'$  is the first outer textual level containing  $C$ ,

and  $i$  is the number of times the function symbol  $F$

has been applied so far.

$\text{compdef}_C(d) =$

if  $d$  is of the form  $V = \varepsilon$ , where  $V$  is a nullary

variable symbol, and  $\varepsilon$  is an expression

in  $\text{lswum}(A)$ , then

$\{V = \text{compexp}_C(\varepsilon)\}$

if  $d$  is of the form  $F(x_0, \dots, x_{n-1}) = \varepsilon$

where  $\varepsilon$  is a simple expression in  $\text{lswum}(A)$ ,

$F$  is an  $n$ -ary variable symbol, and

$x_0, \dots, x_{n-1}$  is the list of formal parameters, then

$$\{ F = \text{compglexp}_C \varepsilon \} \cup \{ \text{compform}_F(x_i) \mid i \in n \}$$

if  $d$  is of the form  $F(x_0, \dots, x_{n-1}) = \varepsilon$

where  $\varepsilon$  is a where-clause expression and of the form

$$\delta \text{ where } E \text{ end}$$

then

$$\{ F = \text{compexp}_E \delta \}$$

$$\cup \{ \text{compform}_F(x_i) \mid i \in n \}$$

$$\cup \{ \text{compdef}_E(d) \mid d \in E \}$$

$$\text{compfor}_F(x) = \{ x = \text{act}(a_0, \dots, a_{m-1}) \}$$

where for each  $i$  in  $m$ ,  $a_i$  is the actual of

the  $i$ 'th invocation of  $F$

$$\text{compglexp}_C(\varepsilon) =$$

if  $\varepsilon$  is of the form  $\text{op}(x_0, \dots, x_{n-1})$

where  $\text{op}$  is an  $n$ -ary constant symbol and

$x_0, \dots, x_{n-1}$  are  $n$  expressions, then

$$\text{op}(\text{compglexp}_C(x_0), \dots, \text{compglexp}_C(x_{n-1}))$$

if  $\varepsilon$  is a nullary variable symbol then

$$\gamma \text{ compvar}_C(\varepsilon)$$

if  $\varepsilon$  is a formal then  $\varepsilon$

if  $\varepsilon$  is of the form  $F(x_0, \dots, x_{n-1})$  where  $F$  is an

$n$ -ary variable symbol and  $x_0, \dots, x_{n-1}$

are  $n$  expressions, then

$$g \wedge (\text{compfunc}_C(F))$$

**Example:** According to the above algorithm, the following program in  $\text{Iswum}(Q)$

```

F(3) where
  F(X) = Y where
    Y = H(X) + H(2) ;
    H(C) = C + G(A) ;
    end ;
  G(B) = A + B ;
  A = 10 ;
end

```

is compiled into

```

result = call0F ;

F = Y ;

Y = call0H + call1H ;

H = C + ggcall0G ;

G =  $\gamma$  A + B ;

A = 10 ;

X = act (3) ;

C = act (X,2) ;

B = act( $\gamma$   $\gamma$  A) ;

```

## 7-2 The Correctness of the Compilation Algorithm:

Informally, we have to prove that, for every extensional algebra  $A$ , the meaning of a program  $P$  in  $\text{Iswum}(A)$  is equal to the meaning of the program  $\text{comprog}(P)$  in  $\text{DE}(\text{Flo}(A))$ . However, there is a slight falsity in this argument because the meaning of a program in  $\text{Iswum}$  is an extensional object, while the meaning of a program in  $\text{DE}$  is intensional. That is, if  $D$  is the domain of the algebra  $A$ , then the value of a program in the source is an element in  $D$ , and in the target is an element in  ${}^U D$  where  $U$  is the universe of  $\text{Flo}(A)$ . To be more precise, we have to prove that the value of a program in  $\text{Iswum}(A)$  is equal to the value of its compilation in  $\text{DE}(\text{Flo}(A))$  at the b-list  $\Lambda$  in  $U$  which is the home world or the origin of the universe.

Furthermore, the meaning of a program in  $\text{Iswum}(A)$  (or in  $\text{DE}(\text{Flo}(A))$  resp.) is dependent on both the algebra  $A$  (or  $\text{Flo}(A)$  resp.) and an environment. However, while an environment for  $\text{Iswum}(A)$  is a function in  $[V \rightarrow D]$ , where  $D$  is the domain of  $A$ ; an environment for  $\text{DE}(\text{Flo}(A))$  is a function in  $[V \rightarrow {}^U D]$ , where  $U$  is the set of b-lists described earlier. Thus, if we are to compare the meaning of a program in  $\text{Iswum}(A)$  with the meaning of a program in  $\text{DE}(\text{Flo}(A))$ , we have to make sure that the environment for the former corresponds to the intensional environment of the latter. That is, the value which the extensional environment assigns to a variable symbol is the same as the value of that symbol in the intensional environment at the b-list  $\Lambda$ . Formally speaking, if  $\sigma$  is the intensional environment then the extensional environment  $\sigma'$  should be defined as  $\lambda v [\sigma(v)(\Lambda)]$ .

We state here the theorem which is the central part of the correctness of the compilation algorithm. However, the proof is too long and technical for this paper, we refer the reader to [Yag83].

**Theorem:**

For every extensional algebra  $A$ , for every intensional  $\text{Flo}(A)$ -environment  $\varepsilon$ , and for every program  $P$  in  $\text{Iswum}(A)$

$$(\models_{\text{Flo}(A), \varepsilon} \text{comprog}(P))(\Lambda) = \models_{A, \varepsilon'} P$$

where  $\varepsilon' = \lambda v [\varepsilon(v)(\Lambda)]$

## 8- Conclusion and Implementation Techniques for DE:

The approach we have described compiles structured first order functional languages into equational nullary order intensional languages. This, we believe, offers a wide range of implementation techniques for functional languages whether on conventional machines or on ones based on data flow principles. In this section we view, briefly, two ways of evaluating programs in DE.

### 8-1 The Reduction Method:

Since the target code is an equational language, a program induces an evaluation tree whose root is **result**, and whose nodes are terms of the intensional algebra  $Flo$ . In this method we consider a program in  $DE(Flo(A))$ , for an extensional algebra  $A$ , as a set of directed equation. This set together with the rewrite rules of the algebra  $Flo$  and those of the algebra  $A$  forms a set of reduction rules for the program. Hence, a reduction on the evaluation tree representing the program. For example, the following  $lswum(Q)$ -program

```
G(3) where
  G(A) = F(2,A) + X ;
  X = 3 + F(Z,6) ;
  F(B,C) = B*B + 2*C ;
  Z = 5 ;
end
```

is translated into the  $DE(Flo(A))$ -program

```
result = call0 G ;

G = gcall0 F + γ X ;

X = 3 + call1 F ;

F = B*B + 2*C ;

Z = 5 ;

A = act (3) ;

B = act (2,Z) ;

C = act (A,6) ;
```

Using the rewrite rules for the algebra  $Flo(Q)$  described before, we can deduce the value of **result** from the above set of DE-equations as follows

$$\begin{aligned}
 \mathbf{result} &= \mathbf{call}_0 G \\
 &= \mathbf{call}_0 (g\mathbf{call}_0 F + \gamma X) \quad (\text{Subs'n}) \\
 &= \mathbf{call}_0 (g\mathbf{call}_0 F) + \mathbf{call}_0 \gamma X \quad (\text{Rule 0}) \\
 &\dots\dots(1)
 \end{aligned}$$

$$\begin{aligned}
 \mathbf{call}_0 \gamma X &= X \quad (\text{Rule 3}) \\
 &= 3 + \mathbf{call}_1 F \quad (\text{Subs'n}) \\
 &= 3 + \mathbf{call}_1 (B*B + 2*C) \quad (\text{Subs'n}) \\
 &= 3 + \mathbf{call}_1 (B*B) + \mathbf{call}_1 (2*C) \quad (\text{Rule 0}) \\
 &= 3 + \mathbf{call}_1 (B) + \mathbf{call}_1 (B) + \mathbf{call}_1 2 + \mathbf{call}_1 C \quad (\text{Rule 0}) \\
 &\dots\dots(2)
 \end{aligned}$$

$$\begin{aligned}
 \mathbf{call}_1 B &= \mathbf{call}_1 (\mathbf{act} (2, Z)) \\
 &= Z \quad (\text{Rule 1}) \\
 &= 5 \quad \dots\dots(3)
 \end{aligned}$$

$$\begin{aligned}
 \mathbf{call}_1 C &= \mathbf{call}_1 (\mathbf{act} (A, 6)) \\
 &= 6 \quad (\text{Rule 1}) \\
 &\dots\dots(4)
 \end{aligned}$$

$$\text{So } \mathbf{call}_0 \gamma X = 3 + 5*5 + 2*6 = 40 \quad (2), (3) \& (4)$$

$$\begin{aligned}
 \mathbf{call}_0 (g\mathbf{call}_0 F) &= \mathbf{call}_0 (g\mathbf{call}_0 (B*B + 2*C)) \\
 &= \mathbf{call}_0 (g\mathbf{call}_0 (B*B)) + \mathbf{call}_0 (g\mathbf{call}_0 (2*C))
 \end{aligned}$$

$$\begin{aligned}
 \mathbf{call}_0 (g\mathbf{call}_0 (B)) &= \mathbf{call}_0 (g\mathbf{call}_0 (\mathbf{act} (2, Z))) \\
 &= \mathbf{call}_0 2 = 2
 \end{aligned}$$



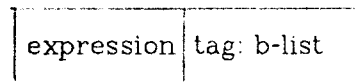
$$\begin{aligned}
\text{call}_0(\text{gcall}_0(C)) &= \text{call}_0(\text{gcall}_0 \text{ act}(A, 6)) \\
&= \text{call}_0 A = \text{call}_0 \text{ act}(3) \\
&= 3
\end{aligned}$$

$$\text{hence } \text{call}_0(\text{gcall}_0 F) = 2*2 + 2*3 = 10$$

$$\begin{aligned}
\text{and } \text{result} &= \text{call}_0(\text{gcall}_0 F) + \text{call}_0 \gamma X \\
&= 10 + 40 = 50
\end{aligned}$$

### The Demand Driven Method:

In this method we consider the program as a data flow net where the nodes are processing stations and the arcs are communication channels carrying data tokens (datons). Each daton consists of two parts; an expression and a b-list as a tag representing the world in the universe at which the expression has to be evaluated.



a daton (tagged expression)

The data flow model we are proposing here is a demand driven one. In such a model a demand is generated from the output port of the net and travels upwards. If such a demand passes through the node  $\sim \boxed{+} \sim$  representing addition say, then it splits into two demands. Each demand travels upwards along the input ports of the node. Thus, we can say that for any expressions  $X, Y$  and a tag  $\alpha$

$$\text{Dem}(X+Y, \alpha) = \text{Dem}(X, \alpha) + \text{Dem}(Y, \alpha)$$

Clearly, if  $C$  is a constant, then

$$\text{Dem}(C, \alpha) = C$$

Basically, there are two classes of processing stations. The first class are the nodes which correspond to the operators of the object (extensional) algebra, e.g the operators of  $Q$  in  $Flo(Q)$ . The second corresponds to the intensional operators of  $Flo$ , e.g  $\gamma$ ,  $call_i$ , ...etc. A node (a processor) of the first class performs operations on the expression part of the daton and needs the tags of its inputs (if there are more than one) to be matching before performing any operation. For example,

if  $\boxed{X \mid \alpha}$  and  $\boxed{Y \mid \beta}$  are two datons

then the processor  $\boxed{+}$  representing the addition operation in  $Q$  fires if and only if  $\alpha = \beta$ . The result is then the new daton

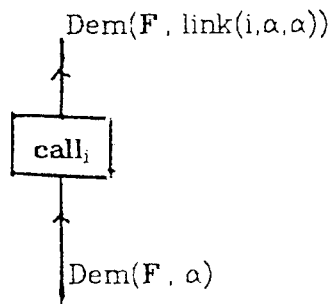
$$\boxed{X+Y \mid \alpha}$$

On the other hand, a node representing an intensional operation manipulates the tag part of the datons. For example, if a demand for the daton

$$\boxed{F \mid \alpha}$$

passes through the node representing the intensional operator  $call_i$ , then the result is a demand for the daton

$$\boxed{F \mid \text{link}(i, \alpha, \alpha)}$$



Clearly, from the definition of the algebra  $Flo$  we can specify the meta-operator  $Dem$  in the following directed equations:

$$Dem(\gamma X, \alpha) = Dem(X, \text{stl}(\alpha))$$

$$\text{Dem}(\text{act}(X_0, \dots, X_{n-1}), \alpha) = \text{Dem}(X(\text{hd}(\alpha)), \text{dtl}(\alpha))$$

$$\text{Dem}(\text{call}_i X, \alpha) = \text{Dem}(X, \beta)$$

$$\text{where } \beta = \text{link}(i, \alpha, \alpha)$$

$$\text{Dem}(\text{gcall}_i X, \alpha) = \text{Dem}(X, \beta)$$

$$\text{where } \beta = \text{link}(i, \alpha, \text{stl}(\alpha))$$

$$\text{Dem}(\text{ggcall}_i X, \alpha) = \text{Dem}(X, \beta)$$

$$\text{where } \beta = \text{link}(i, \alpha, \text{stl}(\text{stl}(\alpha)))$$

**Example :**

We consider here the same program discussed in the last section

**result** = **call**<sub>0</sub> **G** ;

**G** = **gcall**<sub>0</sub> **F** +  $\gamma$  **X** ;

**X** = **3** + **call**<sub>1</sub> **F** ;

**F** = **B\*B** + **2\*C** ;

**Z** = **5** ;

**A** = **act** (**3**) ;

**B** = **act** (**2,Z**) ;

**C** = **act** (**A,6**) ;

As we have mentioned earlier, the value of the program is the value of **result** at the empty b-list  $\Lambda$ . Hence, we start with the demand

$$\begin{aligned} \text{Dem}(\text{result}, \Lambda) &= \text{Dem}(\text{call}_0 \text{G}, \Lambda) \\ &= \text{Dem}(\text{G}, \text{link}(0, \Lambda, \Lambda)) \\ &= \text{Dem}(\text{gcall}_0 \text{F} + \gamma \text{X}, \text{link}(0, \Lambda, \Lambda)) \\ &= \text{Dem}(\text{gcall}_0 \text{F}, \text{link}(0, \Lambda, \Lambda)) + \text{Dem}(\gamma \text{X}, \text{link}(0, \Lambda, \Lambda)) \end{aligned}$$

$$\text{Dem}(\text{gcall}_0 \text{F}, \text{link}(0, \Lambda, \Lambda)) = \text{Dem}(\text{F}, \text{link}(0, \beta, \Lambda))$$

$$\text{where } \beta = \text{link}(0, \Lambda, \Lambda)$$

$$= \text{Dem}(\text{B*B} + \text{2*C}, \text{link}(0, \beta, \Lambda))$$

$$\begin{aligned}
&= \text{Dem}(\mathbf{B}, \text{link}(0, \beta, \Lambda)) * \text{Dem}(\mathbf{B}, \text{link}(0, \beta, \Lambda)) \\
&\quad + \text{Dem}(\mathbf{2}, \text{link}(0, \beta, \Lambda)) * \text{Dem}(\mathbf{C}, \text{link}(0, \beta, \Lambda))
\end{aligned}$$

$$\begin{aligned}
\text{Dem}(\mathbf{B}, \text{link}(0, \beta, \Lambda)) &= \text{Dem}(\mathbf{act}(\mathbf{2}, \mathbf{Z}), \text{link}(0, \beta, \Lambda)) \\
&= \text{Dem}(\mathbf{2}, \beta) \\
&= 2
\end{aligned}$$

$$\begin{aligned}
\text{Dem}(\mathbf{C}, \text{link}(0, \beta, \Lambda)) &= \text{Dem}(\mathbf{act}(\mathbf{A}, \mathbf{6}), \text{link}(0, \beta, \Lambda)) \\
&= \text{Dem}(\mathbf{A}, \beta) \\
&= \text{Dem}(\mathbf{act}(\mathbf{3}), \text{link}(0, \Lambda, \Lambda)) \\
&= \text{Dem}(\mathbf{3}, \Lambda) \\
&= 3
\end{aligned}$$

$$\begin{aligned}
\text{Hence } \text{Dem}(\mathbf{gcall}_0 \mathbf{F}, \text{link}(0, \Lambda, \Lambda)) &= 2*2 + 2*3 \\
&= 10
\end{aligned}$$

$$\begin{aligned}
\text{Dem}(\gamma \mathbf{X}, \text{link}(0, \Lambda, \Lambda)) &= \text{Dem}(\mathbf{X}, \Lambda) \\
&= \text{Dem}(\mathbf{3} + \mathbf{call}_1 \mathbf{F}, \Lambda) \\
&= \text{Dem}(\mathbf{3}, \Lambda) + \text{Dem}(\mathbf{call}_1 \mathbf{F}, \Lambda)
\end{aligned}$$

$$\begin{aligned}
\text{Dem}(\mathbf{call}_1 \mathbf{F}, \Lambda) &= \text{Dem}(\mathbf{F}, \text{link}(1, \Lambda, \Lambda)) \\
&= \text{Dem}(\mathbf{B*B} + \mathbf{2*C}, \text{link}(1, \Lambda, \Lambda))
\end{aligned}$$

$$\begin{aligned}
\text{Dem}(\mathbf{B}, \text{link}(1, \Lambda, \Lambda)) &= \text{Dem}(\mathbf{Z}, \Lambda) \\
&= 5
\end{aligned}$$

$$\begin{aligned}
\text{Dem}(\mathbf{C}, \text{link}(1, \Lambda, \Lambda)) &= \text{Dem}(\mathbf{6}, \Lambda) \\
&= 6
\end{aligned}$$

$$\begin{aligned}
\text{Hence } \text{Dem}(\gamma \mathbf{X}, \text{link}(0, \Lambda, \Lambda)) &= 3 + 5*5 + 2*6 \\
&= 40
\end{aligned}$$

$$\text{and } \text{Dem}(\mathbf{result}, \Lambda) = 10 + 40 = 50$$

## References:

- [Car49]: "The Logical Syntax of Languages", R. Carnap,  
International Library for Phil., Psy., and Scientific Method 1949
- [FMY 83]: "The P-Lucid Programming Manual",  
Faustini, Matthews, and Yaghi, Distributed Computing Report 4,  
University of Warwick
- [ADJ78]: "An Initial Algebra Approach to the specification,  
correctness, and Implementation of Abstract Data Types",  
J. Goguen, J. Thatcher, and E. Wagner,  
In Current Trends In programming Methodology, IV,  
Edited by R. Yeh, Prentice Hall Int. 1978
- [Hen80]: "Functional Programming, Application and Implementation",  
P. Henderson, Prentice Hall International 1981
- [Lan66]: "The Next 700 Programming Languages", Peter Landin,  
CACM Number 3, Vol 9, 1966
- [Mon74]: "Formal Philosophy, Selected Papers of R. Montague",  
Edited by R. Thomason, Yale University press, 1974
- [Tur81]: "The Compilation of an Applicative Language to  
Combinatory Logic", D. Turner, Ph.D. Thesis,  
University of Oxford, 1981
- [WASH83]: "Lucid, The Data Flow Language",  
W. Wadge and E. Ashcroft. Academic Press (to be published)
- [Yag83]: "An intensional Implementation Technique for  
Functional Languages", A. Yaghi, Ph.D. Thesis,  
University of Warwick (in preparation)
- [Yag84]: "Higher Order Functions in Lucid", A. Yaghi  
University of Warwick Report (in preparation)